

Basic Select Statement

Introduction

Queries are used to retrieve data from the database. They can be very flexible in what columns they select, what rows they select, and what aggregate calculations they return. Being able to write accurate and efficient queries is a critical skill when working with databases.

Queries are written with the SELECT statement. The simplified syntax for a select statement is:

```
SELECT [ALL | DISTINCT] column_list  
[FROM table_name [, table_name2 [... , table_name]]  
    [INNER JOIN, LEFT OUTER JOIN, RIGHT OUTER JOIN]  
[WHERE clause]  
[GROUP BY clause]  
[HAVING clause]  
[ORDER BY clause]
```

If it looks complex don't worry. All the "clauses" listed don't need to be used when creating a query. However, what clauses you do use **MUST** be in the order shown above (i.e. HAVING **MUST** be after GROUP BY for example and not vice versa). In addition, there are other functionalities beyond the syntax such as the ability to use string, date, and aggregate functions in your queries to retrieve and calculate data to be returned. We will start by looking at simple queries and then adding additional clauses after that.

All the examples can be executed in the IQSchool database so you can see the results.

SIMPLE QUERY

A simple query can select data from one table. The query can select many rows, calculated data, or even hard coded data when needed. Calculated data be mathematical calculations (subtotal * 1.05) or can be calculated string data (firstname + ' ' + lastname). It can even be a combination of them both ('total: ' + subtotal * 1.05). While formatting what you select in your query is not a common practice (e.g. formatting is best left to the application requesting the data) it can be useful sometimes.

Example:

```
Select      FirstName, LastName, City  
From        Student;
```

As you can see this will select the student's FirstName, LastName and city for all the student records. The From clause indicates the table the columns are in that you are selecting. Queries like this are very simple to create and are quite common.

To select the student's name as one column instead of separate first name and last name columns we can concatenate them together as follows.

```
Select      FirstName || ' ' || LastName, City  
From        Student;
```

In PostgreSQL, the || operator will concatenate string data together as well as numeric data. Just remember to add a space between the names otherwise you will get only one long name with FirstName and LastName combined. You can also use the CONCAT function, but each value must be a string datatype.

There is another difference with the query that returns the names separately than the one that returned them as one column. The column in the second query does not have a name. This is a problem. We should give all columns a name. Calculated columns do not have a name because they are calculated from many different columns and values. So, we must give the column a name. In the example, the "AS" keyword is optional.

```
Select      FirstName || ' ' || LastName AS "Student Name", City  
From        Student;
```

In this course you are mandated to ALWAYS have column names.

To select all the student information for all the students could look like this:

```
Select      StudentID, FirstName, LastName, Gender, StreetAddress, City,  
           Province, PostalCode, Birthdate, BalanceOwing  
From        Student;
```

Since we are selecting all the columns in the student table, we could use the '*' operator. The asterisk represents all columns when used in the column list of a query. The same query could look like this:

```
Select      *  
From        Student;
```

However, there are some possible problems by taking the shortcut and not listing all the column names explicitly.

By looking at the second query can you tell what columns are being returned? It is much easier to maintain the query if all the columns are listed.

It is important to understand that the data being returned will be displayed on a user-friendly screen to most users. This could include a desktop application, a web page, or a mobile device. Consider this scenario. 10 years ago, the query was written with an asterisk to return all the student information to be displayed on the user's desktop computer. The program was written to display 10 columns of data. Last week, someone added another 6 columns of confidential data to the table but those are not supposed to be displayed to the user who is using the original program. Depending on how the program was written it may have these results:

1. It may display all the columns (including the new ones which may contain confidential information) when it only should have displayed the original 10 columns that existed when the program was written, and the query created.
2. It may crash the user's program because the program was created to display 10 columns of data and the additional columns now being returned cause an error.
3. The program behaves normally but there are 6 columns of data being returned to the program that it does not use. An inefficient situation.

While it can be argued that by using the asterisk instead of the column list the query will always return ALL the columns from the table. Even if the table

has columns added at a later date and therefore always keeping the query current. This is something to consider but those situations are fewer.

In this course it is mandated that you will NOT use the asterisk in the column list to select all columns. List the columns explicitly.

Where

So far, we have only selected data that includes ALL the records on the table. While this is not uncommon, it is likely that you will not only want to select the columns you want, but also only the records you want. The WHERE clause allows you to limit the records that are returned based on criteria you provide.

```
Select      StudentID, FirstName, LastName, Gender, StreetAddress, City,  
            Province, PostalCode, Birthdate, BalanceOwing  
From        Student  
Where       StudentID = '198933540';
```

This query would return all the student information for the student with StudentID 198933540. The “Where” clause can contain many different expressions to identify the records the query should return.

Search Criteria Operators and Conditions

Sometimes the conditions that must be met in the data for certain records to be returned involve many expressions that may be combined using AND/OR operators or several other ones.

Operators

The equals sign = is used when looking for an exact match

```
Select      BalanceOwing
From        Student
Where       StudentID = ' 199899200';
```

The not equal sign <> or != is used when looking for something that does not match your criteria

```
Select      FirstName || ' ' || LastName "Student Name", BalanceOwing
From        Student
Where       BalanceOwing != 0;
```

Other common operators include <, <=, >, >=

AND: Both expressions must be true for the record to be returned

OR: If either of the expressions are true then the record is returned

```
Select    FirstName || ' ' || LastName "Student Name", BalanceOwing
From      Student
Where     City = 'Edmonton' and
          Province = 'AB';
```

```
Select    FirstName || ' ' || LastName "Student Name", BalanceOwing
From      Student
Where     City = 'Edmonton' or
          Province = 'AB';
```

BETWEEN Returns all records where the search criteria is between two values (inclusive). You can also use NOT BETWEEN to return records where the search criteria is outside the range.

```
Select    FirstName || ' ' || LastName "Student Name", BalanceOwing
From      Student
Where     StudentID between 198933540 and 199966250;
```

IN allows you to list a number of valid search values for a column. You can always code the same query with a compound expression with multiple OR conditions but that can get quite large with many values. You may also use NOT to return everything that is NOT in the list.

```
Select    FirstName || ' ' || LastName "Student Name"
From      Student
Where     StudentID in
          (198933540, 199912010, 200688700, 200978400);
```

LIKE

The Like operator is used when you use wildcards in your search or perform pattern matching. Wildcards are characters that when used with the Like operator allow the wildcards to represent a character or multiple characters. The Like operator is case sensitive in PostgreSQL.

For example, if we wanted to return all the student's names whose names started with the letter 'D' we would use the % wildcard character. The % wildcard means any character and any number of characters.

```
Select    FirstName || ' ' || LastName "Student Name"  
From      Student  
Where     FirstName Like 'D%';
```

It is important to use the like operator when using wildcards. The % would be interpreted as a literal '%' if we used = instead of Like.

The _ (underscore) is a single character wildcard. This would be used when you are looking for any one character in your search criteria. If we wanted the students' names whose first name was 3 characters long and started with 'Ja' we could use this query.

```
Select    FirstName || ' ' || LastName "Student Name"  
From      Student  
Where     FirstName Like 'Ja%';
```

If we had used a % instead of _ we would also return students whose First Name was longer than 3 characters.

They can also be used in combination with each other. Let's return all the Student Names that start with 'D', the third character is 'n' and end in 's'.

```
Select    FirstName || ' ' || LastName "Student Name"  
From      Student  
Where     FirstName Like 'D_n%s';
```

Regular Expressions

You can also perform a search on a range of values using the caret symbol (~). A query to return all the student names that start with A through J could look like this. Each set of square brackets is for one character.

The [A-J] means that the first character must be in that range and can be followed by any number of characters (including zero) after that.

```
Select    FirstName || ' ' || LastName "Student Name"
From      Student
Where     FirstName ~ '[A-J]';
```

You can add another set of brackets to check if the second character is an "a".

```
Select    FirstName || ' ' || LastName "Student Name"
From      Student
Where     FirstName ~ '[A-J][a]';
```

If you want to check if there are any occurrences of the second character, you add a ".*" between the two sets of square brackets.

```
Select    FirstName || ' ' || LastName "Student Name"
From      Student
Where     FirstName ~ '[A-J].*[y]';
```

You can also check for a particular position. Let's select all the student names who have a Postal Code that starts with T through Z and ends with zero to three. The caret (^) looks at the beginning of the string and the dollar sign (\$) looks at the end of the string.

```
Select    FirstName || ' ' || LastName "Student Name", PostalCode
From      Student
Where     PostalCode ~ '^[T-Z].*[0-3]$';
```


ORDER BY

The ORDER BY clause is used to return the results in a certain order. The results can be sorted by one column or more than one column in either descending (DESC) or ascending (ASC) order. When no order is specified, ASC is the default.

There are numerous ways to use ORDER BY.

```
Select      FirstName "First Name", LastName "Last Name"  
From        Student  
Order By    LastName Desc;
```

Returns the Student Names in Descending order by last name (reverse alphabetical).

Returns the Student Names in Descending order by last name using the alias (i.e. name) we gave to the LastName column.

```
Select      FirstName "First Name", LastName "Last Name"  
From        Student  
Order By    "First Name" Asc, "Last Name" Desc;
```

Returns the Student Names in Ascending order of First Name and when duplicate first names exist, it sorts those by Descending order of Last Name.

ASC is the default and is commonly accepted so it is not uncommon to see the Order By clause used for Ascending ordering without ASC. However, you must use DESC if you want the results sorted in Descending order.

ALL and DISTINCT

ALL and DISTINCT can be in a query or with any aggregate function. ALL is the default and usually not explicitly declared.

```
Select    All FirstName, Province  
From      Student;
```

Returns all the Student First Names and their Province. This returns 16 records because there are 16 students.

```
Select    Distinct FirstName, Province  
From      Student;
```

Returns all the unique (Distinct) student First Names and their Province. This returns only 15 records because there are 2 students with a First Name of Joe from Alberta.

When Distinct is used with an aggregate function (i.e. Count) it would count only the unique (i.e. distinct) values.

AGGREGATE FUNCTIONS

Aggregate functions calculate a single value using the data from many records. If you have used functions in Excel such as SUM, AVERAGE, COUNT, etc. then you have used aggregate functions.

The basic syntax for all the aggregate functions is the same:

aggregate function name ([all | distinct] expression)

- All and distinct are optional with All being the default if none is used.
- expression can be any appropriate column name

AVG

The AVG function returns the average of a column of values. It can only be used on numeric columns and records containing null are ignored and not included in the average calculation.

```
Select    AVG(Mark) "Average Mark"  
From      Registration  
Where     CourseID = 'DMIT2015';
```

Returns the average mark from the registration table for Course DMIT1525. Note that aggregate columns are not columns in the database and therefore do not have a name. Remember, all columns must be given a name if they do not have one.

SUM

The SUM function returns the sum of a column of values. It can only be used on numeric columns and records containing null are ignored and not included in the sum calculation.

```
Select    SUM(Amount) "Payment Amount"  
From      Payment  
Where     StudentID = '200495500';
```

Returns the Sum of all the payments for StudentID 200495500.

MIN

The MIN function returns the minimum value from a column of values. It can be used with columns that have a numeric, date or character datatype. Records with a null value in that column are ignored.

```
Select    MIN(Mark) "Lowest Mark"  
From      Registration  
Where     CourseID = 'DMIT2015';
```

Returns the lowest Mark in DMIT2015.

MAX

The MAX function returns the maximum value from a column of values. It can be used with columns that have a numeric, date or character datatype. Records with a null value in that column are ignored.

```
Select    MAX(Mark) "Highest Mark"  
From      Registration  
Where     CourseID = 'DMIT2015';
```

Returns the highest Mark in DMIT2015.

COUNT

The COUNT function returns the number of non-null values from a column of values OR returns the number of records that match the “where” criteria.

```
Select    COUNT (DateReleased) "Number of Staff Released"  
From      Staff;
```

Returns the number of staff that have been released (number of staff that have a value in the DateReleased column).

```
Select    COUNT (*) "Number of Staff Released"  
From      Staff;
```

Returns the number of records returned by this query (number of staff RECORDS in the staff table).

NOTE: The * operator is an overloaded operator. This means that it has more than one meaning depending on how it is used. In a column list in a query, it means all columns (which should be avoided unless necessary). In the count function it means “records”.

STRING FUNCTIONS

There are many string functions that will allow you to work with parts of strings to define search criteria and results.

LENGTH (column | expression)

```
Select Length (FirstName) "First Name Length", FirstName  
From Student;
```

Returns the length of each FirstName and the FirstName of each Student.

```
Select FirstName  
From Student  
Where Length (FirstName) = 3;
```

Returns all the First Names that are 3 characters long.

LEFT (column |expression, length)

```
Select Left (FirstName,3) "First 3 characters"  
From Student;
```

Returns the first 3 characters from the FirstName of each student.

RIGHT (column |expression, length)

```
Select Right (FirstName,3) "Last 3 characters"  
From Student;
```

Returns the last 3 characters from the FirstName of each student.

SUBSTRING (column | expression, start, length)

```
Select Substring (CourseID, 5, 3) "Character 5, 6, 7"  
From Course;
```

For each CourseID, return the 3 characters starting at the 5th character. The first character in an SQL string is considered character one (1).

REVERSE (column | expression)

```
Select Reverse (FirstName) "Backwards Name"  
From Student;
```

Returns the first names in reverse.

UPPER (column | expression)

LOWER (column | expression)

```
Select Upper (FirstName) "Uppercase", Lower (FirstName) "Lowercase"  
From Student;
```

Returns the names in uppercase and/or lowercase.

LTRIM (column | expression)

RTRIM (column | expression)

```
Select Ltrim (Rtrim ('    Hello World    '));
```

Removes the leading (LTRIM) or trailing (RTRIM) spaces from the column data or expression. To remove leading and trailing spaces you must nest one function in another.

DATE Functions

Current_Date returns the system date
Current_Time returns the system time
DatePart (field, date1) returns a numeric representation of the “field”
portion of date1

Field for Date Portions

hour	Hour of Day	0 to 23	
day	Day of Month	1 to 31	(depending on month)
dow	Day of Week	0 to 6	(Sunday to Saturday)
isodow	Day of Week	1 to 7	(ISO numbering) (Monday to Sunday)
doy	Day of Year	1 to 365/366	
week	Week number of the year	1 to 53	
month	Month number of the year	1 to 12	
quarter	Quarter number of the year	1 to 4	
year			
decade			
century			
millennium			

Field for Time Portions

microseconds
milliseconds
second
minute

-- You can also use Fields to get the character representation of dates.

Select To_Char (Current_Date, 'DAY');

Returns the day of the week (i.e. MONDAY, TUESDAY, etc.). Note that the “Field” is case sensitive: “Day” would return “Monday”.

GROUP BY

The GROUP BY clause is used in conjunction with aggregate functions to provide subtotal calculations of the aggregate function. We could easily return the average Mark from the grade table. What if we wanted the average Mark for each Course? For each student? For each course for each student? To do this we would use GROUP BY.

```
Select AVG(Mark)"Average Mark"  
From Registration;
```

Returns the average of all the marks in the Registration table.

```
Select CourseID, AVG(Mark)"Average Mark"  
From Registration  
Group BY CourseID;
```

Returns the average mark for each CourseID (GROUP BY CourseID) in the Registration Table. Another way to think of the syntax GROUP BY is the 'for each'.

It is important to note that you cannot combine an aggregate column with a non-aggregate column(s) unless you group by all the non-aggregate columns in your query.

```
Select CourseID, AVG(Mark)"Average Mark"  
From Registration;
```

```
ERROR: column "registration.courseid" must appear in the GROUP BY clause or  
be used in an aggregate function
```

Remember that an aggregate returns one calculated value. How can it return the average mark for the Registration table (one value) as well as each course ID that is in the Registration table (many values). It simply does not make sense and cannot be done. So the error you see above reminds you that you must group by all the columns you are using in your query that are not part of the aggregate function. In this case, CourseID.

HAVING

Having is like the where clause. Except, it applies its criteria after group by has grouped its aggregate data. With the GROUP BY clause, we selected the average mark of CourseID. What if we only wanted to return the CourseIDs and average mark for courses that have averages that are greater than 80?

```
Select CourseID, AVG(Mark)"Average Mark"  
From Registration  
Group BY CourseID  
Where AVG(Mark) > 80;
```

ERROR: syntax error at or near "Where"

Fails because the where clause must be before the Group By Clause. Remember, a query does not need to use all the clauses but they must be in the right order!

```
Select CourseID, AVG(Mark)"Average Mark"  
From Registration  
Where AVG(Mark) > 80  
Group BY CourseID;
```

ERROR: aggregate functions are not allowed in WHERE

Fails because and aggregate cannot be in a where clause. It also does not really make sense. It is not possible to return only the CourseID's that have average over 80 until all the Course averages have been calculated. And that is done by the Group By Clause. So, the Group By Clause must execute before the criteria is applied to return only the CourseID's with averages over 80.

```
Select CourseID, AVG(Mark)"Average Mark"  
From Registration  
Group BY CourseID  
Having AVG(Mark) > 80;
```

Works!The Select with the group by selects the averages for each course and then the having clause returns only the ones whose average was over 80.

What if we only wanted to return the CourseID's that have averages over 80 and did not want the actual average returned. Simply do not select the AVG(Mark). Remember, what we select (return to the user) does not have to match columns used in the where or having clauses.

```
Select CourseID  
From Registration  
Group BY CourseID  
Having AVG(Mark) > 80;
```

Works!

Having is used with aggregate functions only. It is used to apply where clause functionality AFTER the aggregate function and group by has occurred. Only use Having in this situation. For queries without a group by clause you must use the Where clause.

VIEWS

A view is a query that is stored in the database. It behaves very much like a virtual table that was created by selecting rows and columns from other tables. In most situations it can be used just like an actual table. There is no real overhead with creating a view, as there is no actual data stored in the view. It is created by select statement(s) and the data remains in the original tables.

Some common uses for SQL views are:

- simplifying data retrieval from complex queries
- hiding the underlying table structure
- controlling access to data for different users

SIMPLIFYING DATA RETRIEVAL

As you may be starting to notice, sometimes we need to code complicated select statements containing joins, aggregates, function, unions, etc. to produce the results we want. A view can simplify this task. Some or all of a complex select statement can be created as a view and future queries of that information can be executed against the view.

For example, if we were often executing queries involving publishers and their titles, we would constantly be writing selects with a join to retrieve data from those tables and then executing additional criteria in the where clause to retrieve more specific data. A view could be created which returns all the publishers and their titles. Any specific queries about that data could then be executed against the view. This would allow us to not have to code a join condition every time we wanted information from those tables together as we have already created a view that contains that information.

HIDING THE UNDERLYING TABLE STRUCTURE

By creating a view, we create an additional layer between the user and the tables. If the definition of a table changes only the view needs to be altered to accommodate the changes. As long as the view contains the same data as it did before the changes to the underlying table(s), then the user is unaware of the changes, and they are of no concern to them.

A common example of this is when a table gets too big and some of the records are moved to an archive table. If a view was being used that contained all the rows and records of the original table it could now be altered to select all the records from the original table and UNION them with the records from the archive table. As far as the user of the view is concerned, the view contains the same records as before. The definition of the view had to be changed to select the records from both the original table and the archive table, but the result is the same. The view contains all the records that were there originally.

CONTROLLING ACCESS TO DATA

By creating a view, you can restrict which columns certain users are able to see. Suppose that some information in an employee table should be available to everyone, but some columns are restricted information. A view can be created that contains the columns that are available to everyone, and permissions can be granted on that view to everyone. Other views could contain other combinations of columns and permissions that could be granted accordingly to those views for the appropriate users and groups.

DATA MODIFICATION

A user can do more than just select data from views. As mentioned before a view can be treated like a table (with a few exceptions). You can update, insert, and delete records in a view just as if you were doing so on an actual table. Any changes to the view are reflected in the records in the table(s) that make up the view.

Here are some points to keep in mind when modifying records through a view:

- Modifying records through a view must meet all rules and constraints that were created on the table(s) that the view is based on
- You cannot insert, update, or delete from a view that is based on more than one table
- If the view select statement contains distinct, group by, having, or union, you cannot modify data in that view

CREATING A VIEW

SYNTAX:

```
CREATE [OR REPLACE] VIEW viewname
```

```
AS
```

```
Select ...
```

NOTE:

- a view cannot reference more than 1600 columns (depending on column types)

Create View StudentCourseView

As

```
Select Student.FirstName, Student.LastName, Course.CourseID,  
        Course.CourseName  
From Student  
        Inner Join Registration  
            On Student.StudentID = Registration.StudentID  
        Inner Join Course  
            On Course.CourseID = Registration.CourseID;
```

What can we do with this view?

```
Select FirstName, LastName, CourseName  
From StudentCourseView;
```

Works!

Insert into StudentCourseView

(FirstName, LastName, CourseID, CourseName)

Values

('Bob', 'Smith', 'SDEV1201', 'New Database Course');

Fails. You cannot insert data through a view if the view is based on multiple tables.

ERROR: cannot insert into view "studentcourseview"

Views that do not select from a single table or view are not automatically updatable.

SQL state: 55000 Detail: Views that do not select from a single table or view are not automatically updatable. Hint: To enable inserting into the view, provide an INSTEAD OF INSERT trigger or an unconditional ON INSERT DO INSTEAD rule.

Update StudentCourseView

Set CourseName = 'ChangedCourseName'

Where CourseID = 'DMIT1508';

Fails. You cannot Update through a view if the view is based on more than one table.

ERROR: cannot update view "studentcourseview"

Views that do not select from a single table or view are not automatically updatable.

SQL state: 55000 Detail: Views that do not select from a single table or view are not automatically updatable. Hint: To enable updating the view, provide an INSTEAD OF UPDATE trigger or an unconditional ON UPDATE DO INSTEAD rule.

Delete From StudentCourseView
Where CourseID = 'DMIT1508';

Fails. You cannot Delete through a view if the view is based on more than one table.

ERROR: cannot delete from view "studentcourseview"

Views that do not select from a single table or view are not automatically updatable.

SQL state: 55000 Detail: Views that do not select from a single table or view are not automatically updatable. Hint: To enable deleting from the view, provide an INSTEAD OF DELETE trigger or an unconditional ON DELETE DO INSTEAD rule.

DROPPING VIEWS

Views can be dropped using the DROP keyword.

SYNTAX

DROP VIEW [IF EXISTS] *viewname*;